Appears in *Workshop on Foundations of Computer Security*, 2009.

Post-Hoc Separation Policy Analysis with Graph Algorithms

Lee Pike

Galois, Inc.
leepike@galois.com

Abstract. We present graph algorithms for analyzing potential information flow in systems, modeled as labeled directed graphs, particularly focusing on discovering and analyzing separation policies that require minimal changes to an existing system. We have implemented the algorithms in an open-source publicly-available tool.

1 Introduction

Under ideal conditions, long before a system is implemented or even architected, its security policy would be formulated. Ideally, the policy would be an instance of a well-studied security policy, such as the Bell-La Padula model [1] or the Biba model [2]. The system would be designed from the outset to respect the policy so that the kinds, location, and direction of information flows are well-specified. These can be considered to be *prescriptive* security models that prescribe the conditions a system must satisfy to guarantee that it has such-and-such a property.

However, often security considerations come after a system has been designed. For example, consider the development of Security-Enhanced Linux [3]. Originally, Linux was not built to be used in security-critical environments (indeed, it originally was not even conceived for commercial use), but as its popularity grew, so did its security-critical uses, which motivated the development of SELinux's policy engine. One consequence of post-hoc security "add-ons" is that doing so increases the complexity (and reduces the confidence in) the system; for example, the reference SELinux security policy is around one-quarter million lines.

Contributions and Scope In this paper, we address the difficulties associated with post-hoc security analyses by describing an intuitive and practical model for discovering and improving information flow properties of a system and discovering "high-risk" components and information flows. We particularly focus on separation properties. The mathematical basis of our model uses directed graphs in which edges are partially-ordered. This is a simple model of information flow known in the folklore of security modeling. Nevertheless, these analyses have not been presented together in a coherent form. We have found them to be a nice way to gain an initial understanding of the security properties of a system and to provide heuristics for modifying systems to enhance their security. The open-source tool we are releasing in conjunction with this paper provides easy access to these ideas.

We do not address the problem of information encoding in which data of type T is transformed—often maliciously—into data of type T' in order to transmit it over channels that permit data of type T' but not T to be transmitted. The intent of our approach and tool is for preliminary analyses of potential information flow in system design.

We also do not address the problem of how to ensure that a graph model accurately represents information flows of a system.

Outline The remainder of this paper proceeds as follows. In Section 2, we present the basic mathematical notation, definitions, and results upon which the analysis is based. In Section 3, we describe integrity and confidentiality analyses. In Section 4, we describe how a separation policy can be defined for our model and we give algorithms for taking an arbitrary graph and transforming it into one in which the policy is satisfied. We put together the analyses presented in the paper to work through an extended example in Section 5. A brief description of the tool we have implemented to aid in this analysis is described in Section 6. We briefly describe related work in Section 7, and concluding remarks are given in Section 8.

2 Definitions and Notation

This section describes the mathematical model upon which we base our analysis. We provide some motivation for the interpretation of the model, but we save most motivating examples for Section 3.

We model distributed systems as *directed labeled graphs*. Let $(V, L, \rightarrow)$ be a directed labeled graph where V is a finite set of vertices, L is a finite set of edge labels, and $\rightarrow \subseteq L \times V \times V$ is a directed-edge relation between two vertices that has a nonempty *set of labels* associated with it. (Later, we describe algorithms that remove labels from edges; if a label-set becomes empty, the edge is removed.) If a label $l \in X$ is in the label-set associated with an edge (X, v, v'), we say that (X, v, v') *contains* l or that the edge is *labeled* l, realizing that labels are non-unique. We sometimes represent an edge (X, v, v') containing the label l and pointing from vertex v to vertex v' by $v \xrightarrow{l} v'$.

Vertices model entities that create, receive, and distribute information (e.g., computers, virtual machines, files, processes, threads, ports, etc.); information may flow from or to vertices. Labels model the type of the information flow between entities (e.g., shared bits, shared pages, firewalls with rules, ports, etc.). Edges model channels between entities, including the direction of the information flow.

Let $\leq$ be a partial-order on edge labels (note that this generalizes weighted graphs, in which weights are totally ordered). The $\leq$ relation models an order on the potential information flow: if $l \leq m$, then any information that can be passed over an edge labeled l can be passed over an edge labeled m. If $l \leq m$, we say that m *dominates* l or that the *information type* of l is *subsumed by* the information type of m.

For example, if the channels represent types in a programming language, and the types are not abused by encoding information, then while a natural number satisfies an integer type (channel), a negative integer does not satisfy a natural number type (channel). Two channels might be incomparable: for example, in a strongly-typed software

system in which threads communicate via shared variables, the set of variables over which integers can be shared and the set of variables over which strings can shared may be disjoint.

This model does not address semantic transformations a vertex might make on data from one type to another type, which may or may not be malicious. The model is therefore most useful for either analyzing unintended non-malicious information flows in systems that in which data does not undergo numerous transformations, or as a first-approximation model to be refined.

In the remainder of this paper, we make two "canonical-form" assumptions regarding the graphs we consider (we state, without proof, that the algorithms presented in this paper preserver the assumptions):

1. Self information-flow is always possible (i.e., the edge relation $\rightarrow$ is always reflexive), so for all vertices v and all labels l, we assume $v \xrightarrow{l} v$.
2. If an edge contains the label l and label $m \leq l$, then the edge also contains m.

For convenience, we omit reflexive edges from the graphs we display or define in this paper. Furthermore, we may omit an edge label that is dominated by another label, since it is implied.

Definition 1 (Transitive Closure). *The* transitive closure *of a vertex v with respect to a label l in the graph G is the set of vertices to which information of type l can flow from v. That is, it is the smallest set satisfying the following equation:*

$$T_l^v = \left\{ v_i \in V \;\middle|\; \begin{array}{l} v \xrightarrow{l} v_i, \text{ or there exists } v_j \in T_l^v \\ \text{such that } v_j \xrightarrow{l} v_i \end{array} \right\}$$

We sometimes write $v \xrightarrow{l}_* v'$ to abbreviate $v' \in T_l^v$. If $v \xrightarrow{l}_* v'$, we say that v' is *reachable from v with respect to l* or that v' *is l-reachable from v*.

Analogously, we can define transitive closures up to some depth k:

Definition 2 (k-Transitive Closure). *the* k-transitive closure *of a vertex v with respect to a label l is the set of edges reachable from v with respect to l, in k or fewer edges, where k is a natural number. That is,*

$$T_l^v(k) = \left\{ v_i \in V \;\middle|\; \begin{array}{c} k = 0 : v_i = v \\ \text{otherwise} : Q(v_i,\, l,\, k) \end{array} \right\}$$

where $Q(v_i,\, l,\, k) =$ there exists some $v_j \in T_l^v(k-1)$ and $v_j \xrightarrow{l} v_i$.

So the 0-transitive closure of a vertex is itself, the 1-transitive closure of a vertex is itself and all the vertices reachable from it in one step, the 2-transitive closure of a vertex is itself and all the vertices reachable in two or fewer steps, and so on.

To compute a transitive closure or k-transitive closure, at most each edge in the in the graph needs to be followed. If $|\rightarrow|$ is the number of edges in the graph, an algorithm for computing the (k-) transitive closure is $\mathcal{O}(|\rightarrow|)$.

For graph $(V,\ L,\ \rightarrow)$, vertices $v,\ v' \in V$ are *partitioned* with respect to l if and only if both $v \xrightarrow{l}_* v'$ and $v' \xrightarrow{l}_* v$ are false. The sets of vertices $V,\ V' \subseteq V$ are partitioned with respect to l if every $v \in V$ and every $v' \in V'$ are partitioned. Vertices $v,\ v' \in V$ are *k-partitioned* with respect to l if both $v' \in T_l^v(k)$ and $v \in T_l^{v'}(k)$ are false. A *cut* with respect to a partition (or k-partition) is a set of labels such that if they are removed from a graph, the partition (or k-partition) is satisfied.

The *transpose* of a labeled directed graph $G = (V,\ L,\ \rightarrow)$ is the graph $G^T = (V,\ L,\ \leftarrow)$ such that $v \xleftarrow{l} v'$ in graph G^T if and only if $v' \xrightarrow{l} v$ in graph G.

3 Confidentiality and Integrity Analysis

In this section, we build on the preliminaries developed in Section 2 for security analyses. Three essential security properties are the "CIA" properties: confidentiality, integrity, and availability. Informally, confidentiality is about preventing information from leaking from an entity, and integrity is about preventing data or objects from being manipulated by incoming data. Availability is about services or data being present for some entity when needed.

Our model of systems as labeled directed graphs deals with all three properties at a relatively high level of abstraction. In particular, we do not model data explicitly. Rather, the model deals with the kind of channel or bandwidth of a channel over which entities transmit or receive data. Thus, availability is modeled by which entities share directed edges and what their labels are. If data is to pass through multiple channels, the transitive closure of an entity models availability.

While the graph itself models availability, we present various confidentiality and integrity analyses based on taking the transitive closure of a graph (in Section 4, we present algorithms that remove channels (i.e., labels) so that a graph satisfies a post-hoc separation policy; availability becomes an issue when channels are removed). We focus on confidentiality first, in Section 3.1 below. We present definitions and then some prove some claims about confidentiality, and in Section 3.2, we do likewise for integrity. These analyses can be thought of as ways to abstract potentially complex graphs to quickly make security-relevant judgments about them.

These analyses form the basis of the separation policy analysis described in Section 4.

3.1 Confidentiality Analysis

In our model, the concept of confidentiality captures what information can be sent where. From a single vertex, this concept is captured simply by taking the transitive closure (Definition 1) of that vertex with respect to some label. Thus, the confidentiality closure CC_l^v of vertex v with respect to label l is the transitive closure of v with respect to l (we introduce new terminology just to emphasize that we are examining *confidentiality* using transitive closures). Likewise, the *k-confidentiality-closure* $CC_l^v(k)$ is defined as the k-transitive closure.

Now, let CC_l be the set of all confidentiality closures with respect to l: $CC_l = \{CC_l^v \mid v \in V\}$. The *maximum confidentiality closures with respect to label l* is the set

of all confidentiality closures with respect to label l such that no other confidentiality closure is a strict superset. That is, $CC_l^{max} = \{s \mid s \in CC_l \text{ and for all } s' \in CC_l, s' \not\supset s\}$.

Intuitively, maximum confidentiality closures abstract the greatest extent to which information can be distributed. In a system in which information flow is tightly controlled, for example, one expects to see a large number of small closures rather than a small number of large closures.

Proposition 1, states that if maximum confidentiality closures do not intersect, then they represent a partitioning of the graph:

Proposition 1 (Partitioning). *Suppose that for any two sets X, $Y \in CC_l^{max}$, $X \neq Y$ implies $X \cap Y = \emptyset$. Then for any $x \in X$ and $y \in Y$, y is not reachable from x with respect to l; that is, $x \xrightarrow{l}_* y$ is false.*

Proof. By assumption, no vertex x is in more than one set X in the maximum confidentiality closure CC_l^{max}. By the definition of a confidentiality closure, $CC_l^x \subseteq X$, and by assumption, $y \notin X$. Thus, there exists no $y \neq x$ such that $y \notin X$ and $x \xrightarrow{l}_* y$. □

Dually to maximum confidentiality closures, the *minimum confidentiality closures with respect to label l* is the set of confidentiality closures with respect to label l such that no other confidentiality closure is a strict subset. That is,

$$CC_l^{min} = \{s \mid s \in CC_l \text{ and for all } s' \in CC_l, s' \not\subset s\}$$

Intuitively, minimum confidentiality closures are "data sinks". For set $X \in CC_l^{min}$, any data sent to any vertex in X can only be sent to other domains in X.[1]

Finally, we define *k-maximum confidentiality closures with respect to label l* and *k-minimum confidentiality closures with respect to label l*. First, let $CC_l(k)$ be the set of all k-confidentiality closures with respect to l: $CC_l(k) = \{CC_l^v(k) \mid v \in V\}$. The former can be defined as

$$CC_k^{max}(k) = \{s \mid s \in CC_l(k) \text{ and for all } s' \in CC_l(k), s' \not\supset s\}$$

and the latter as

$$CC_k^{min}(k) = \{s \mid s \in CC_l(k) \text{ and for all } s' \in CC_l(k), s' \not\subset s\}$$

3.2 Integrity Analysis

An integrity analysis builds on the same definitions used for the confidentiality analysis. Here, we are taking the perspective that integrity is about what information flows can *reach* a node as opposed to confidentiality, which is about which information flows can

[1] A reviewer asked if minimum confidentiality closures are equivalent to the set of *strongly connected components* (SCCs) of a directed graph. The SCCs are a partitioning of the graph such for each two vertices v and v' in a partition, there is a path from v to v'. As a counterexample, consider the graph on the left in Figure 1: it has five SCCs, but only four minimum confidentiality closures.

leave a node. So mathematically, we apply the same analyses but to the transpose G^T of the graph G.

The *integrity closure IC_l^v of vertex v with respect to label l* is the transitive closure of v with respect to l in graph G^T. The integrity closure of vertex v is the set of vertices that can potentially send data dominated by the information type l, to v. The *k-integrity closure IC_l^v of vertex v with respect to label l* is the k-transitive closure of v with respect to l in graph G^T.

The following proposition shows that if for some vertex v its confidentiality and integrity closures are equal, then the confidentiality and integrity closures for every vertex reachable from v are equal.

Proposition 2 (Transitively-Closed Graph). *Suppose that for vertex v, $CC_l^v = IC_l^v$. Then for every $v' \in CC_l^v$, $CC_l^{v'} = IC_l^{v'}$.*

Proof. We prove that $CC_l^{v'} = CC_l^v$ and that $IC_l^{v'} = IC_l^v$. To prove $CC_l^{v'} = CC_l^v$, we show that each is a subset of the other. $CC_l^{v'} \subseteq CC_l^v$ since $v' \in CC_l^{v'}$, by assumption. To show that $CC_l^v \subseteq CC_l^{v'}$, consider that $v' \in IC_l^v$, so $v' \xrightarrow{l}_* v$, by the definition of integrity closure, so v is reachable from v' with respect to l, so every vertex reachable from v is also reachable from v'.

We have shown that $CC_l^{v'} = CC_l^v$. The proof that $IC_l^{v'} = IC_l^v$ is analogous. □

Like for confidentiality closures, we also define maximum and minimum integrity closures. The *maximum integrity closures with respect to label l* over graph G is equal to the maximum *confidentiality* closures with respect to label l over graph G^T. Similarly, the *minimum integrity closures with respect to label l* over graph G is equal to the minimum *confidentiality* closures with respect to label l over graph G^T.

Intuitively, maximum integrity closures represent the greatest extent to which information can be collected by a single vertex. For a given information type, each closure shows the extent to which a single vertex may collection information from other vertices. Conversely, minimum integrity closures are intuitively "data sources". For a set $X \in IC_l^{min}$, no information within X originates from outside X.

The intuitive differences between the notions of maximum and minimum confidentiality closures can be captured by the following example:

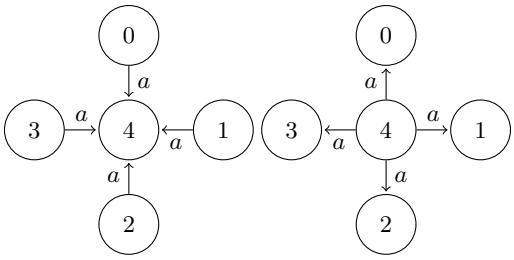

Fig. 1. Confidentiality and Integrity Closure Examples

Example 1 (Closures Example). Consider the two graphs with five nodes each in Figure 1. There is only one edge label a. The graph on the right is the transpose of the graph on the left. We give the maximum and minimum confidentiality and integrity closures for the graph on the left in the left column and on the right in the right column:

$CC_a^{max} =$	$CC_a^{max} = \{\{0, 1, 2, 3, 4\}\}$
$\{\{0, 4\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\}$	$CC_a^{min} = \{\{0\}, \{1\}, \{2\}, \{3\}\}$
$CC_a^{min} = \{\{4\}\}$	$IC_a^{max} =$
$IC_a^{max} = \{\{0, 1, 2, 3, 4\}\}$	$\{\{0, 4\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\}$
$IC_a^{min} = \{\{0\}, \{1\}, \{2\}, \{3\}\}$	$IC_a^{min} = \{\{4\}\}$

For each graph, the four closures give a different abstraction of it. As expected, the confidentiality closures in the left graph are equal to the integrity closures in the right graph, since they are transposes of one another.

4 Separation Policies

The model of distributed systems we have presented thus far allows us to analyze the potential unencoded information flows between entities in a system. However, suppose we wished to stipulate that certain information flows should not occur between various entities. A separation policy formally states which information flows should be disallowed.

However, because we are doing post-hoc analysis, a system may not have been designed from the outset to respect the policy. In this case, we may wish to modify the system so that it does. In particular, we may wish to eliminate (or reduce the bandwidth of the information type) certain channels that results in the policy being violated.

In this section, we present the notion of a separation policy, and we describe algorithms[2] for analyzing a distributed system with respect to a separation policy and for eliminating channels so that the policy is satisfied. We begin by mentioning two very simple algorithms, which run in constant time, that simply disconnect the source vertex (the source of information) or the sink vertex (the vertex to which information flows), respectively. A natural criterion for eliminating channels is to eliminate the minimum number of channels required to satisfy the policy; as we will see, doing so is equivalent to the hitting-set problem [4], and this problem is NP-hard. That said, we describe a greedy algorithm that approximates the optimal solution in polynomial time. Finally, we discuss heuristics for discovering separation policies.

We begin by defining a separation policy, both for paths of arbitrary length and for paths bounded by a length k. Intuitively, a separation policy specifies which vertices should have no information flow possible (up to some information type) between them. We stipulate that separation policies are antireflexive: $v \overset{l}{\nrightarrow} v \notin \nrightarrow$, and $v \overset{l}{\nrightarrow}_k v \notin \nrightarrow$ for all v and k.

[2] Technically, these are not algorithms in the sense that we do not define concrete data representations. Complexity analysises of these "algorithms" assume canonical efficent data representations found in the literature [4].

Definition 3 (Separation Policy). *A* separation policy *is a relation* $\nrightarrow \subseteq L \times V \times V$. *A graph* $(V, L, \rightarrow)$ respects *the separation policy if and only if for each policy element* $v \overset{l}{\nrightarrow} v'$, $v \overset{l}{\rightarrow}_* v'$ *does not hold.*

A k-separation policy *is a relation* $\nrightarrow \subseteq \mathcal{N} \times L \times V \times V$ *(we use the same relation symbol* $\nrightarrow$ *for both separation and* k*-separation policies; context distinguishes their uses). A graph* $(V, L, \rightarrow)$ respects *the* k*-separation policy if and only if for each policy element* $v \overset{l}{\nrightarrow}_k v'$, $v' \in T_l^v(k)$ *does not hold.*

For a policy element $v \overset{l}{\nrightarrow} v'$ *or* $v \overset{l}{\nrightarrow}_k v'$, v *is the* source *and* v' *is the* sink.

What use is a k-separation policy? For small values of k, a k-separation policy is intuitive. For example, violations of a 2-separation policy imply that either information can flow directly from vertex v to v' or that there is an intermediary, v'', such that information can flow from v to v'' and from v'' to v'. If the threat model for a system states that no more than one vertex might be compromised, then we must ensure that no vertex can serve as an intermediary between two portions of the system that should remain unconnected.

The following example describes how a particular well-known confidentiality policy might be stated in our model. In addition, we generalize the policy slightly by parameterizing it by the type of information flow.

Example 2 (Parameterized Bell-Lapadula). Informally, the Bell-Lapadula security model partitions the set of vertices V into *security levels* and defines a total order on the partitions [1]. Then, if partition P is greater than partition P' under the ordering, and if $v \in P$ and $v' \in P'$, then v is not allowed to *write down* to v', and v' is not allowed to *read up* to v. In our model, we can represent reads with one set of edge labels, where each label in the set denotes a particular type of data reading, and we can represent writes with another set of edge labels and stipulate that any label in the one set is incomparable with a label from the other.

As noted in Section 2, the model does not capture potential semantic modifications vertices might make to data.

To determine if a distributed system, modeled as a labeled directed graph, satisfies a separation policy, we check the confidentiality closures for each policy element.

Algorithm 1 (Policy Satisfaction Algorithm). For each policy element $(l, v, v') \in \nrightarrow$,
1. Compute the confidentiality closure CC_l^v.
2. If $v' \in CC_l^v$, then the system does not satisfy the policy.

Otherwise, the policy is satisfied.

Similarly, to determine if a distributed system satisfies a k-separation policy, we examine the k-transitive closures.

Suppose that some graph does not satisfy a separation policy. We would like to construct a subgraph that does satisfy the policy. The key to doing so is for every policy element, remove some edge label along each path from the source to the sink. This prevents information flow along the path. Of course, we wish to minimize the number of edge labels removed so as to not prevent allowed information flow. We present

successfully more sophisticated algorithms for doing so. In the following, we present algorithms for separation policies; algorithms for k-separation policies are analogous.

From a practical perspective, a system designer takes the output of these algorithms as advisory: "if one wishes to satisfy such-and-such a policy, one needs to reduce the information flow of this set of channels." Doing so, however, might require an untenable refactoring of the system or prevent necessary information flow. In such cases, the system designer might decide to do one or more of the following:

- Factor the security policy.
- Document the security weakness.
- Incorporate security protections *within* vertices that violate a policy. For example, if the graph is interconnected machines (real or virtual), the designer might use a system that supports mandatory access control (e.g., SELinux) to ensure tighter control on information flow within the vertex.

For each algorithm, it is correct if it (1) terminates, (2) returns a graph that respects the policy (*soundness*), and (3) under some metric, a minimum number of edge labels are removed (*precision*). For each algorithm presented, the proofs of these are either straightforward or we can reduce them to proofs for known algorithms.

Our first two algorithms break paths by removing edge labels at the end or beginning of paths, respectively. That is, we remove edge labels to the sink or from the source. (We omit the analogous algorithms for k-separation policies.

The first algorithm takes a graph $(V,\, L,\, \rightarrow)$ and returns a subgraph in which edge labels to the sink violating the policy are removed:

Algorithm 2 (The Sink Separation Algorithm).

1. For each policy element $(l,\, v,\, v') \in \nrightarrow$, let
$$R_{(l,\, v,\, v')} = \rightarrow \setminus \left\{ v_i \xrightarrow{l} v' \mid v_i \in CC_l^v \right\}$$
2. Return the graph $(V,\, L,\, \bigcap R_{(l,\, v,\, v')})$.

For Algorithm 2, the metric for "minimum edge labels removed" is that the for each policy element, the confidentiality closure of the source is maximized while satisfying the policy.

Algorithm 3 is the dual in the sense that it transforms a graph into one respecting a separation policy by removing edges from the source vertex (i.e., v for the separation policy element $(l,\, v,\, v')$).

Algorithm 3 (The Source Separation Algorithm).

1. For each policy element $(l,\, v,\, v') \in \nrightarrow$, let
$$R_{(l,\, v,\, v')} = \rightarrow \setminus \left\{ v \xrightarrow{l} v_i \;\middle|\; v_i \in CC_l^{v_i} \right\}$$
2. Return the graph $(V,\, L,\, \bigcap R_{(l,\, v,\, v')})$.

For Algorithm 3, the metric for "minimum edge labels removed" is that the for each policy element, the integrity closure of the sink is as large as possible while satisfying the policy.

The complexity of Algorithms 2 and 3 is the complexity of computing an integrity closure or confidentiality closure, respectively (linear in the number of edges: $\mathcal{O}(|\rightarrow|)$), multiplied by the number of policy elements, $|\nrightarrow|$, so the total complexity is $\mathcal{O}(|\rightarrow| \cdot |\nrightarrow|)$.

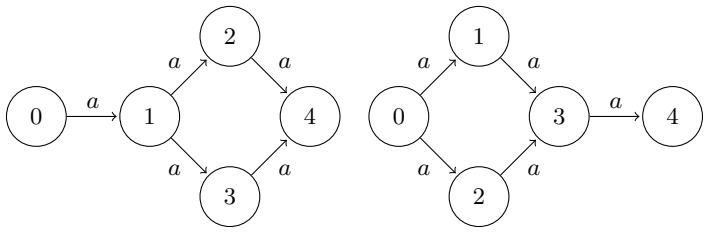

Fig. 2. Algorithms 2 and 3 Fail to Remove the Minimum Number of Edges

Neither Algorithm 2 nor Algorithm 3 removes the minimum number of edge labels from a graph to satisfy a separation policy. For example, consider the two graphs in Figure 2. In the graph on the left, Algorithm 2 fails to remove the minimum number of edges required to separate 0 and 4, and for the graph on the right, Algorithm 3 fails to remove the minimum number of edges to separate 0 and 4.

Our final algorithm computes a minimal cut (i.e., removes the minimum number of edge labels possible) to satisfy the separation policy. We solve the problem by reducing it to the *hitting set* problem. In the hitting set problem, we have a collection of subsets $S_0, S_1, \ldots, S_k$ of a universe

$$U = \bigcup_{0 \leq i \leq k} S_i$$

(the union of each S_i) of elements. The problem is to find the smallest hitting set $H \subseteq U$ such that for each $0 \leq i \leq k$, $S_i \cap H \neq \emptyset$. That is, some element from each S_i is an element of H.

Our problem of finding a minimum-cost set of cuts to satisfy a separation policy is reducible to the hitting set problem as follows. Represent paths in a graph as sets of labeled edges. For each policy element (l, v, v'), let $S_{(l, v, v')}$ be the collection of all paths from v to v' with respect to l, and let S be the union of each of these collections. To cut each path, we need to remove an edge label from each path; thus, to find a minimum set (the set returned is not necessarily unique) of edge labels to remove, we solve the hitting set problem.

The hitting problem is NP-hard. However, a greedy approximation algorithm exists, the error of which is bounded [4]. The idea is to iteratively choose the most prominent element until a hitting set is complete. Initialize the hitting set to be the empty set

($H := \emptyset$), and initialize $S' := S$, where $S = \{S_0, S_1, \ldots, S_k\}$, the collection of subsets we wish to hit. Then,

Algorithm 4 (Greedy Algorithm for Hitting Set).

1. If S is the empty set, then stop.
2. Otherwise, choose some element e such that the number of sets S_i where $e \in S_i$ is maximized.
3. $H := H \cup \{e\}$.
4. $S := S \setminus \{S_i \mid e \in S_i\}$.
5. Go to Step 1.

The running time is polynomial in the size of S and U. A bound on the difference between the hitting set returned by Algorithm 4 and the optimal solution is $ln(j) + 1$, where j is the size of the largest S_i in S [4].

Another solution using a greedy algorithm is to formulate the problem as an instance of the min-cut problem over network flow graphs, which has been well-studied in graph theory and combinatorial optimization [5]. The basic insight of this formulation is to assign weights to edges on paths violating a policy element. Edges used by a greater number of paths get assigned a higher weight. Iteratively, edges with the highest weights are removed.

4.1 Heuristics for Discovering Separation Policies

In imposing a security policy on a system, sometimes it is useful to discover where the system's "joints" are. That is, are there a few edges responsible for a large degree of information flow through the system? These joints are probable targets for shoring up security, either by decomposing them into less permissive channels or imposing additional security mechanisms. In this section, we describe some heuristics for discovering locations in (graphs of) systems in which separation policies might be enforced (the heuristics described in this section have not been implemented in the tool described in Section 6).

First, we describe the heuristic of searching for cliques within a graph. A *clique with respect to* l is a set of vertices C such that for every two vertices $v, v' \in C$ and $v \neq v'$, $v \xrightarrow{m} v'$, where $l \leq m$. Finding a maximal clique in a graph is a NP-complete problem [4]. An algorithm to find a maximal clique can be modified to find all cliques by finding a maximal clique, removing it from the graph, finding a maximal clique in the remaining subgraph, and so on.

A clique strongly suggests that its vertices have needed information flow. However, vertices in a clique may nevertheless share edges with vertices outside the clique; if a system designer wishes to separate the two cliques, she may execute one of the above algorithms to determine which edge labels to remove to separate them.

Dually, *independent sets* (or *stable sets*) are sets of vertices in a graph such that no two vertices in an independent set shares an edge (we generalize this to an *independent set with respect to* l, where no two edges in the independent set share an edge labeled by l or a label $m \geq l$. Vertices in an independent set are candidate vertices to be separated.

Another heuristic builds on maximum confidentiality and integrity closures. Maximum closures are the greatest fix-points of information flow. If two vertices v and v' do not appear in the same maximum confidentiality or integrity closure, than either v is unreachable from v', or vice versa; there is no imposition on the system if they are henceforth required to remain separated.

Finally, another possibility for discovering possible separation policies is to compute a *graph partition* [6]. A graph partition partitions the vertices into disjoint sets (usually, there is a requirement that the subsets be of nearly equal size) such that the edges between vertices in disjoint subsets be minimized. The graph partitioning problem is NP-complete and is applicable to other domains, such as workload balancing.

5 Putting it Together: Extended Example

We have presented a number of information flow analyses; in this section, we work through a small example to get a feel for how to put them together in a security analysis.

Suppose we have a set of processes that interact through shared access to a central file system under a discretionary access control model. Processes that have permissions on files may transfer those permissions to other processes.

Let us consider three possible actions that a process x may take with a process y:

- $read(x,\, y)$: process x provides process y with read permission to a subset of files to which x has read permission.
- $write(x,\, y)$: process x provides process y with write permission to a subset of files to which x has write permission.
- $read - write(x,\, y)$: process x provides both read and write permission a subset of files to which x has read and write permission.

We can order these actions. In this model, information flow is measured in terms of which processes can read and write to which files. The $read()$ and $write()$ actions are incomparable, but $read() \leq read - write()$ and $write() \leq read - write()$.

Consider the graph in Figure 3. Vertices are processes. Edges denote actions processes perform on each other. The labels r, w, and rw correspond to the actions $read()$, $write()$, and $read - write()$, respectively. Thus, $x \xrightarrow{r} y$ denotes $read(x,\, y)$ (note that because $read() \leq read - write()$, if $x \xrightarrow{rw} y$, then $x \xrightarrow{r} y$).

Additionally, suppose we begin with the following separation policy: we begin by stipulating that if a process x can provide another process y read permissions on a file, then y cannot provide x with any write permissions: that is, for edge $x \xrightarrow{r} y$, there is a corresponding $x \overset{w}{\not\to} y$, where $x \neq y$.

To begin our analysis, we compute the maximum confidentiality closures for each action:

- $read()$: $\{\{A,\, C,\, D,\, E,\, F,\, G,\, H,\, I\},\, \{B\}\}$
- $write()$: $\{\{A,\, B,\, C,\, D,\, E,\, F,\, G,\, H\},\, \{A,\, B,\, F,\, H,\, I\}\}$
- $read - write()$: $\{\{A,\, F,\, H,\, I\},\, \{B\},\, \{C,\, D,\, G\},\, \{C,\, E,\, G\}\}$

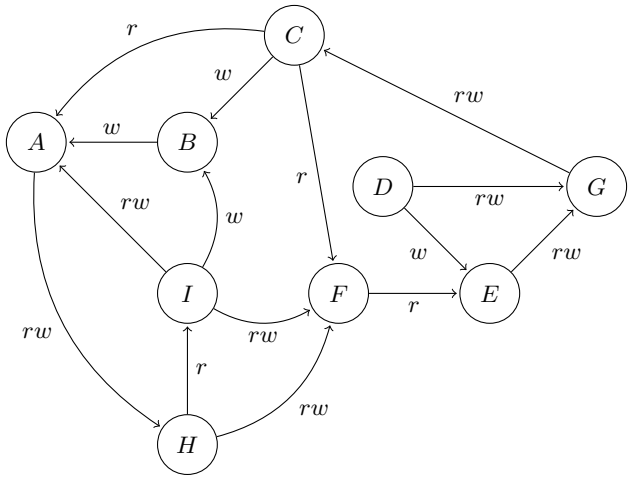

Fig. 3. Process Permission Graph

From this we learn the some facts. For example, B cannot obtain read permissions from any other process, but some process can provide every process but B with read permissions. Because all of the elements of the second $write()$ closure listed belong to the first one too except for element I, we know the second closure is I's confidentiality closure.

Now let us look at the some "data sinks" by computing the minimum confidentiality closures:

- $read()$: $\{\{A, C, E, F, G, H, I\}, \{B\}\}$
- $write()$: $\{\{F\}\}$

From these we see that no other process can provide B with read permissions (which we already knew from taking the maximum confidentiality closures) and that every process can provide F write permissions. Notice that D is missing from minimum confidentiality closures for read permissions; this tells us that its confidentiality closure is equal to the first closure of the maximum confidentiality closures. Taking D's confidentiality closure explicitly confirms this: $\{A, C, D, E, F, G, H, I\}$. So let us see how many steps it takes D to transfer read permissions to every process in D's closure. $CC_r^D(4) = \{A, C, D, E, F, G, H\}$, so D can read every process but I in four steps. But $CC_r^D(1) = \{D, G\}$, so D can only reach G in one step.

Now that we have a sense of some of the information flow properties of the graph, we turn our attention to the stated separation policy. We execute each of the three separation algorithms defined in Section 4 to see what, if any, edge labels should be removed to satisfy the policy defined above. Interestingly, the first two algorithms remove exactly the same edges, while the third algorithm removes many eight fewer edges:

- Algorithm 2 (remove sink edge labels) and Algorithm 3 (remove source edge labels): $A \overset{rw}{\rightarrow} H$, $E \overset{rw}{\rightarrow} G$, $G \overset{rw}{\rightarrow} C$, $H \overset{rw}{\rightarrow} F$, $I \overset{rw}{\rightarrow} A$, $I \overset{rw}{\rightarrow} F$, $C \overset{r}{\rightarrow} A$, $C \overset{r}{\rightarrow} F$, $F \overset{r}{\rightarrow} E$, and $H \overset{r}{\rightarrow} I$.
- Algorithm 4 (greedy hitting set): $A \overset{rw}{\rightarrow} H$, and $E \overset{rw}{\rightarrow} G$.

So satisfying our separation policy requires modifying only two channels!

6 Implementation

The model, definitions, and algorithms in this paper have been implemented in a small utility that is freely available under a BSD3 license.[3] The utility is written in the pure functional language Haskell [7]. Accompanying the code are sample input files. We call the tool the *Graph Abstractor Toolkit* (GAT).

The modeler begins by defining a graph in a small Haskell module that is imported by the analysis module. Haskell is typed, so types are defined for vertices and labels, respectively. The graph is represented as a set of pairs of the form $(l, (v, v'))$, where l is an edge label, and the pair (v, v') denotes an edge from v to v'. A partial order over labels is optionally given as well as a separation policy.

The user then interacts with the program at the command line by function calls, which are documented, corresponding to each of the analysises presented herein. For example, (`integClosureK` v l k) returns the k-integrity closure of vertex v with respect to label l to depth k, and (`sepPolicySourceEdges`) returns the result of Algorithm 3, the edge labels from the source vertex to be removed to satisfy a separation policy.

7 Related Work

Zdancewic briefly overviews the history of information-flow analysis for security [8]. Much of the research has focused on noninterference and language-based information-flow techniques [9], particularly focusing on type-enforcement, or program analysis [10]. Our model is more appropriate for analyzing complex systems in which there is no precise noninterference policy or for doing analyses for *discovering* potential separation policies. A number of papers have addressed security models generally [11–13], and work to formally verify security policies has been done [14].

8 Discussion

This work can be extended in a variety of ways. For example, some of these analyses can be generalized to *hypergraphs*, a generalization of graphs in which an edge can connect subsets of vertices. Hypergraphs are a way to express complex systems as recursive graphs, useful for modeling complex systems hierarchically. Similarly, one could attempt to combine information flow analyses with *statecharts*, a formalism for modeling complex reactive systems [15].

[3] `http://www.cs.indiana.edu/~lepike/pub_pages/infoflow.html`

In our model, channels are static. For some dynamic systems, the model could be generalized to handle dynamic channels. Such a model would combine a state machine with an information flow graph in which channels are introduced or removed based on the state of the system.

Although we have presented a set of tools herein for the working systems engineer, these might be combined and automated to provide higher-level analyses or automate some of the inferences made based on the analyses. Indeed, one could imagine the interpretations of the analyses we have presented herein being crafted as heuristics,that together with domain-specific rules, guide a system designer to building secure systems.

Acknowledgments

Don Stewart helped uncover some performance issues in the Graph Abstractor Toolkit. Iavor Diatchki and Joe Hurd commented on a draft of this paper. Our anonymous referees provided valuable feedback.

References

1. Bell, D.E.: Looking back at the Bell-La Padula model. In: ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference, IEEE (2005) 337–351
2. Biba, K.J.: Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation (April 1977)
3. McCarty, B.: SELinux: NSA's Open Source Security Enhanced Linux. O'Reilly (2004)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press (1990)
5. Manber, U.: Introduction to Algorithms: A Creative Approach. Addison-Wesley (1989)
6. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company (1979)
7. Jones, S.P., (editors), J.H.: Haskell 98: A non-strict, purely functional language. Technical report (February 1999)
8. Zdancewic, S.: Challenges for information-flow security. In: Proceedings of the 1st International Workshop on Programming Language Interference and Dependence (PLID04)
9. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21** (2003) 2003
10. Bergeretti, J.F., Carré, B.A.: Information-flow and data-flow analysis of while-programs. ACM Transactions on Programmng Languages and Systems **7**(1) (1985) 37–61
11. Landwehr, C.E., Heitmeyer, C.L., Mclean, J.: A security model for military message systems. ACM Transactions on Computer Systems **2** (1984) 198–222
12. Millen, J.K.: 20 years of covert channel modeling and analysis. In: IEEE Symposium on Security and Privacy. (1999) 113–114
13. McLean, J.: Security models. In Marciniak, J., ed.: Encyclopedia of Software Engineering. John Wiley & Sons (1994)
14. Guttman, J.D., Herzog, A.L., Ramsdell, J.D., Skorupka, C.W.: Verifying information flow goals in security-enhanced linux. Journal of Computer Security **13** (2004) 2005
15. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3) (1987) 231–274